6.001 SICP
Variations on a Scheme

- Scheme Evaluator – A Grand Tour
  - Making the environment model concrete
  - Defining eval defines the language
    - Provides mechanism for unwinding abstractions
- Techniques for language design:
  - Interpretation: eval/apply
  - Semantics vs. syntax
  - Syntactic transformations
- Beyond Scheme – designing language variants
  - Lexical scoping vs. Dynamic scoping

Building up a language...
1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

Stages of an interpreter
Lexical analyzer
Parser
Evaluator
Environment
Printer

The Core Evaluator
1. eval/apply core

Meval
(define (meval exp env)
  (cond
    ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp env))
    ((quoted? exp) (text-of-quotation exp))
    ((assignment? exp) (eval-assignment exp env))
    ((definition? exp) (eval-definition exp env))
    ((if? exp) (eval-if exp env))
    ((lambda? exp) (make-procedure (lambda-parameters exp)
                                 (lambda-body exp) env))
    ((begin? exp) (eval-sequence (begin-actions exp) env))
    ((cond? exp) (eval (cond->if exp) env))
    ((application? exp) (mapply (meval (operator exp) env)
                                (list-of-values (operands exp) env)))
    (else (error "Unknown expression type -- EVAL" exp)))))

Basic Semantics: m-eval & m-apply
- primitive expressions
  - self-evaluating, quoted
- variables and the environment
  - variable definition, lookup, and assignment
- conditionals
  - if, cond
- procedure creation
- sequences
  - Begin
- procedure application
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (m-eval (first-operand exps) env)
                    (list-of-values (rest-operands exps) env)))))

(define (mapply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure) arguments
                              (procedure-environment procedure)))
        (else (error "Unknown procedure type -- APPLY" procedure))))

Side comment – procedure body

• The procedure body is a sequence of one or more expressions:

(define (foo x)
  (do-something (+ x 1))
  (* x 5))

• In m-apply, we eval-sequence the procedure body.
**Syntax Abstraction**

- Semantics
  - What the language means
  - Model of computation
- Syntax
  - Particulars of writing expressions
  - E.g. how to signal different expressions
  - Separation of syntax and semantics: allows one to easily alter syntax

---

**Basic Syntax**

```scheme
(define (tagged-list? Exp tag)
  (and (pair? Exp) (eq? (car exp) tag)))

(define (if? exp) (tagged-list? exp 'if))
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (application? exp) (pair? exp))

(define (operator app) (car app))
(define (operands app) (cdr app))

(define (no-operands? args) (null? args))
(define (first-operand args) (car args))
(define (rest-operands args) (cdr args))
```

---

**Example – Changing Syntax**

- Suppose you wanted a "verbose" application syntax, i.e., instead of
  
  ```scheme
  (proc arg1 arg2 ...) 
  ```
  
  use
  
  ```scheme
  (CALL proc ARGS arg1 arg2 ...)
  ```

- Changes – only in the syntax routines!

```scheme
(define (application? exp) (tagged-list? exp 'CALL))
(define (operator app) (cadr app))
(define (operands app) (cdddr app))
```

---

**Implementing "Syntactic Sugar"**

- Idea:
  - Easy way to add alternative/convenient syntax
  - Implement a simpler "core" in the evaluator

- "let" as sugared procedure application:

```scheme
(let ((<name1> <val1>)
      (<name2> <val2>))
  <body>)
```

```scheme
(lambda (name1 name2) <body>)
```

---

**Detect and Transform the Alternative Syntax**

```scheme
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp)
          (lookup-variable-value exp env))
        ((quoted? exp)
          (text-of-quotation exp))
        ...)
```

```scheme
(define (m-apply (operator exp) env)
  (m-eval (list-of-values (operands exp) env))
  (else (error "Unknown expression" exp)))
```
Let Syntax Transformation
FROM
(let ((x 23)
      (y 15))
  (dosomething x y))
TO
  (lambda (x y) (dosomething x y))

Define Procedures
(define foo (lambda (x) <body>))
(define (foo x) <body>)

Semantic implementation – just another define:
(define (eval-definition exp env)
 (define-variable! (definition-variable exp)
  (m-eval (definition-value exp) env))

Syntactic transformation:
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp) ; formal params
                   (cddr exp))))) ; body

How the Environment Works
• Abstractly – in our environment diagrams:
  x: 10
  plus: (procedure ...)

  list of variables
  list of values

  frame

  E2

  E1

  E3

• Concretely – our implementation (as in SICP)

Extending the Environment
• (extend-environment '(x y) (list 4 5) E2)

  Abstractly

  Concretely

Scanning” the environment
• Look for a variable in the environment...
  • Look for a variable in a frame...
    • Loop through the list of vars and list of vals in parallel
    • Detect if the variable is found in the frame
  • If not found in frame (out of variables in the frame),
    look in enclosing environment
The Initial (Global) Environment

- **setup-environment**
  - (define (setup-environment)
    (let ((initial-env (extend-environment (primitive-procedure-names)
                                           (primitive-procedure-objects)
                                           the-empty-environment)))
     (define-variable! 'true #T initial-env)
     (define-variable! 'false #F initial-env)
     initial-env))

- define initial variables we always want
- bind explicit set of "primitive procedures"
  - here: use underlying scheme
  - in other interpreters: assembly code, hardware, ....

Variations on a Scheme

- More (not-so) stupid syntactic tricks
  - LetSeq:
    (letseq ((x 4) (y (+ x 1))) . . . )
  - Infix notation:
    (+(* 4 3)7) ➔ ((4 * 3)+ 7)

- Semantic variations
  - _Lexical vs dynamic scoping_
    - Lexical: defined by the text
    - Dynamic: defined by the runtime behavior

Diving in Deeper: Lexical Scope

- Why?
  - our semantic rules for procedure application!
  - "hang a new frame"
  - "bind parameters to actual args in new frame"
  - "evaluate body in this new environment"

Read-Eval-Print Loop

- (define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (m-eval input the-global-env)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

Diving in Deeper: Lexical Scope

- How does our evaluator achieve lexical scoping?
  - environment chaining
  - procedures that capture their lexical environment
- make-procedure:
  - stores away the evaluation environment of lambda
  - the "evaluation environment" is always the enclosing lexical scope

Lexical Scope & Environment Diagram

Will always evaluate (+ x y z) in a new environment inside the surrounding lexical environment.
**Alternative Model: Dynamic Scoping**

- **Dynamic scope:**
  - Look up free variables in the caller's environment rather than the surrounding lexical environment

- **Example:**

```scheme
(define (pooh x)
  (bear 20))

(define (bear y)
  (+ x y))

(pooh 9) => 29
```

**Dynamic Scope & Environment Diagram**

Will evaluate \(+ x y\) in an environment that extends the caller's environment.

---

**A "Dynamic" Scheme**

```scheme
(define (m-eval exp env)
  (cond
   ((self-evaluating? exp) exp)
   ((variable? exp) (lookup-variable-value exp env))
   ...)
   ((lambda? exp)
    (make-procedure (lambda-parameters exp)
    (lambda-body exp)
    "no-environment") ;CHANGE: no env
    ...
   ((application? exp)
    (d-apply (m-eval (operator exp) env)
    (list-of-values (operands exp) env)
    env) ;CHANGE: add env
    (else (error "Unknown expression -- M-EVAL" exp)))))
```

**A "Dynamic" Scheme – d-apply**

```scheme
(define (d-apply procedure arguments calling-env)
  (cond
   ((primitive-procedure? procedure)
    (apply-primitive-procedure procedure arguments))
   ((compound-procedure? procedure)
    (eval-sequence
     (procedure-body procedure)
     (extend-environment
      (procedure-parameters procedure)
      arguments
      calling-env)) ;CHANGE: use calling env
    (else (error "Unknown procedure" procedure)))))
```

---

**Summary**

- Scheme Evaluator – Know it Inside & Out
- Techniques for language design:
  - Interpretation: eval/apply
  - Semantics vs. syntax
  - Syntactic transformations
- Able to design new language variants!
  - Lexical scoping vs. Dynamic scoping